ISI Research Report
ISI/RR-89-241
June 1990

4

University of Southern California

Rivi Sherman
Amir Pnueli

Model Checking for Linear Temporal Logic: An Efficient Implementation

INFORMATION SCIENCES INSTITUTE

213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT This document is approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ISI/RR-89-241	5. MONITORING ORGANIZATION REPORT NUMBER(S) -------
6a. NAME OF PERFORMING ORGANIZATION USC/Information Sciences Institute 6b. OFFICE SYMBOL (*If applicable*)	7a. NAME OF MONITORING ORGANIZATION -------
6c. ADDRESS (*City, State, and ZIP Code*) 4676 Admiralty Way Marina del Rey, CA 90292-6695	7b. ADDRESS (*City, State, and ZIP Code*) -------
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA RADC NASA-Ames 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER RADC: F30602-88-C-0135, DARPA Order No. 6131 NASA-Ames: NCC-2-539
8c. ADDRESS (*City, State, and ZIP Code*) --over--	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. ------- PROJECT NO. ------- TASK NO. ------- WORK UNIT ACCESSION NO. -------

11. TITLE (*Include Security Classification*)
Model Checking for Linear Temporal Logic: An Efficient Implementation (Unclassified)

12. PERSONAL AUTHOR(S) S' rman, Rivi; Pnueli, Amir

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Research Report	FROM _____ TO _____	1990, June	

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP
09	02	

18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
~~linear-temporal logic~~, model-checking algorithms, program verification.

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

Program verification is a critical problem in computer science. Many believe that manual, informal verification is acceptably reliable and more practical than formal verfication. However, verification of distributed programs is difficult to achieve manually.

This report provides evidence to support the claim that model checking for linear temporal logic (LTL) is "practically efficient." ~~We describe~~ two implementations of a linear temporal logic model checker. One is based on transforming the model checking problem into a satisfiability problem; the other checks an LTL formula for a finite model by computing the cross-product of the finite state transition graph of the program with a structure containing all possible models for the property. ~~We~~ experimented with a set of mutual exclusion algorithms and tested safety and liveness under fairness for these algorithms.

We provide the syntax and semantics of LTL, a detailed example of a finite state concurrent program, and express safety and liveness-under-fairness properties for this program in LTL. Both implementations are based on the tableau algorithm, which is described in detail. Finally, we discuss the basic ideas behind the two different model-checking algoirthms, and provide more details regarding the implementations and experimental results.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (*Include Area Code*)	22c. OFFICE SYMBOL
Victor Brown Sheila Coyazo	213/822-1511	

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

8c. (continued)

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, VA 22209

Air Force Systems Command (DARPA)
Rome Air Development Center
Griffiss Air Force Base, NY 13441

NASA-Ames
Moffett Field, CA 94035

ISI Research Report
ISI/RR-89-241
June 1990

University of Southern California

Rivi Sherman
Amir Pnueli

Model Checking for Linear Temporal Logic: An Efficient Implementation

INFORMATION SCIENCES INSTITUTE

213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

This research was sponsored in part by the National Aeronautics and Space Administration (NASA-Ames) under Cooperative Agreement NCC-2-539 and in part by the Rome Air Development Center of the Air Force Systems Command under Contract No. F30602-88-C-0135, DARPA Order No. 6131. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of NASA, RADC, DARPA, the U.S. Government, or any person or agency connected with them.

1 Introduction

1.1 Is automatic program verification important?

Program verification is a critical problem in computer science. Many believe that manual, informal verification is acceptably reliable and more practical than automatic formal verification. However, verification of distributed programs is difficult to achieve manually.

Consider the following simple distributed algorithm. P_1 and P_2 are two processes that are executed on a processor. They share a common variable t, and each has a local variable $y_i, i = 1, 2$ that can be read by the other process. We assume that at each step exactly one process is active, i.e., it executes the current statement. P_1 and P_2 share a common resource (printer or disk) and the algorithm is meant to guarantee that at most one process uses the resource at a given time. We say that the process is in the *critical section* when it uses the resource. L_3 and M_3 are the critical sections for P_1 and P_2, respectively.

The reader is challenged to verify whether the following algorithm satisfies two properties:

1. "Mutual exclusion" is guaranteed: it is never true that P_1 is at L_3 and P_2 is at M_3.
2. Liveness is guaranteed: always if P_1 (P_2) is at L_1 (M_1) (requesting the resource), then eventually P_1 (P_2) will be at L_3 (M_3) (getting the resource).

DECLARE $t : [0..1];$
INITIALLY $t = 1;$

PROCESS P_1

DECLARE $y_1 : [0..1];$
INITIALLY $y_1 = 0;$
L_0 : goto L_0; | $\{t := 1;$ goto $L_1;\}$

$L_1 : y_1 := 1;$

L_2 : if $(y_2 = 0 \vee t = 0)$ goto $L_3;$
if $(y_2 = 1 \wedge t = 1)$ goto $L_2;$

critical section $L_3 : y_1 := 0;$ goto $L_0;$
END
||
PROCESS P_2

DECLARE $y_2 : [0..1];$
INITIALLY $y_2 = 0;$

M_0 : goto M_0; | $\{t := 0$; goto $M_1;\}$

$M_1 : y_2 := 1;$

M_2 : if $(y_1 = 0 \vee t = 1)$ goto M_3;
if $(y_1 = 1 \wedge t = 0)$ goto M_2;

critical section $M_3 : y_2 := 0$; goto M_0;
END

1.2 How to verify it automatically

Program verification is defined as follows: given an implementation A and a specification ψ, does A satisfy ψ ? Putting this into more formal terms, in the most general case, program verification is equivalent to the problem of checking if $L_1 \subseteq L_2$, where L_1, L_2 are two languages defined by Turing machines. Hence, in the most general case the verification problem is undecidable.

Programs, however, have been verified manually. Manual verification is sometimes misleading and almost always very tedious, especially in the case of parallel or distributed programs. Some systems designed to support formal, manual verification have been developed in the last few years. For example, Crawford and Goldschlag provide an interactive theorem prover to support the verification of distributed systems ([CG87]). Theorem provers provide only a partial and most often not satisfactory answer to the problem.

Taking a different approach, Clarke et al. [CES83] suggested that by focusing strictly on finite state programs, one could provide a fully automatic verifier that would still be applicable to such domains as communcation protocols and cicuit design. In [CES83] an algorithm for checking a finite state model against a property specified by Computation Tree Logic (CTL) is provided. The algorithm is linear in the size of the model and the property. The algorithm was implemented [B86, BC86], and was proved to be useful for verifying circuit design. However, the CTL formalism is not always enough to express properties of distributed systems [L80, EH86]. Linear Temporal Logic (LTL), on the other hand, seems to provide the required expressive power at the cost of having a model checking problem, which is NP-complete [SC82].

Thus, the linear temporal logic advocates claim to have the required expressive power while those in favor of branching time claim to be "efficient". In [LP85] an $O(|M|2^{|P|})$ algorithm for checking a finite state model M against a linear temporal logic formula p is described. It is claimed that since the property is usually small and the worst case rarely happens, the algorithm is "practically efficient." Branching Time "struck back" in [EL85], where it was shown that any model checking algorithm for LTL implies an algorithm of the same complexity for CTL*, the extended version of branching time temporal logic,

which subsumes both LTL and CTL.

In this work, we provide evidence to support the claim that model checking for LTL is "practically efficient."

We describe two implementations of a linear temporal logic model checker. One is based on transforming the model checking problem into a satisfiability problem. The other checks an LTL formula for a finite model by computing the cross-product of the finite state transition graph of the program with a structure containing all possible models for the property. We experimented with a set of mutual exclusion algorithms and tested safety and liveness under fairness for these algorithms. We believe that the measurements we have done for these examples provide experimental evidence for the practicality of model checking of linear temporal logic formulae.

Section 2 provides the syntax and semantics of linear temporal logic. Section 3 provides a detailed example of a finite state concurrent program and expresses safety and liveness-under-fairness properties for this program, in LTL. Both implementations are based on the *tableau* algorithm described in Section 4. Section 5 discusses the basic ideas behind the two different model checking algorithms. Section 6 provides more details regarding the implementations and experimental results.

2 Linear temporal logic

A *temporal logic formula* is defined over a set Φ_0 of atomic formulae, using the boolean operators $\vee$ and $\neg$ and the temporal operators *next* ($\bigcirc$) and *until* ($\mathcal{U}$). A *model* for a temporal formula p is an infinite sequence of states $\sigma : s_0, s_1, \ldots$, and a mapping $\tau : \{s_i \mid i \geq 0\} \rightarrow 2^{\Phi_0}$ assigning to each state s_i the set of atomic formulae that are true at this state.

Φ denotes the set of all temporal formulae that are inductively constructed from Φ_0 as follows:

if $P \in \Phi_0$ then $P \in \Phi$

if $p, q \in \Phi$ then $\neg p$ and $p \vee q \in \Phi$

if $p, q \in \Phi$ then $\bigcirc p \in \Phi$ and $p \,\mathcal{U} q \in \Phi$

For a given model σ and a temporal formula p, we say that (σ, j) *satisfies* p, denoted by $(\sigma, j) \models p$, if p is evaluated to *true* on the jth state of σ.

Formally:

$(\sigma, j) \models P$, for $P \in \Phi_0$ iff P is evaluated to true in s_j by the mapping τ, that is, if $P \in \tau(s_j)$

$(\sigma, j) \models \neg p$ iff $(\sigma, j) \not\models p$

$(\sigma, j) \models p \vee q$ iff $(\sigma, j) \models p$ or $(\sigma, j) \models q$

$(\sigma, j) \models \bigcirc p$ iff $(\sigma, j+1) \models p$

$(\sigma, j) \models p \,\mathcal{U} q$ iff there exists $i \geq j$ such that $(\sigma, i) \models q$ and for all k, $j \leq k < i$, $(\sigma, k) \models p$

We use the following operators as abbreviations:

and: $p \wedge q = \neg(\neg p \vee \neg q)$

implies: $p \rightarrow q = \neg p \vee q$

equivalent: $p \equiv q = (p \rightarrow q) \wedge (q \rightarrow p)$

eventually: $\Diamond p = \mathbf{T}\mathcal{U} p$, $(\sigma, j) \models \Diamond p$ if for some $i \geq j, (\sigma, i) \models p$.

always: $\Box p = \neg\Diamond\neg p$, $(\sigma, j) \models \Box p$ if for all $i \geq j, (\sigma, i) \models p$.

A model σ satisfies p if $(\sigma, 0)$ satisfies p.

A formula p is *satisfiable* if there exists a model that satisfies p.

A formula p is *valid* if for every model σ, σ satisfies p. Hence, a formula p is valid iff $\neg p$ is not satisfiable.

The temporal formalism is used to specify properties of finite state programs. A program is defined as the set of all possible computations. A computation of a given program is a sequence of states, starting from the initial state, where each state is defined by an assignment of values to all program variables. A computation can be viewed as a model for a temporal logic formula. A program satisfies a property p, if for every computation σ of this program, σ satisfies p.

For linear temporal logic, the *model checking problem* is stated as follows: given a finite state transition graph $M = (N, E, r)$, where r is a special root node (the initial state), a mapping function $\pi : N \rightarrow 2^{\Phi_0}$ assigning atomic propositions to states, and a linear temporal logic formula p, does every path of M initiated at r satisfy p?

3 Example

To demonstrate the terms and notation, we give an example of a distributed program and some properties that this program is required to satisfy.

We distinguish between two types of program properties, *safety* and *liveness*. Safety properties state that nothing "bad" happens throughout the computation, while liveness properties state that something "good" eventually will happen during the computation.

We assume an asynchronous semantics, meaning that the set of all program computations consists of all possible interleavings of process computations.

Peterson's Mutual Exclusion Algorithm

DECLARE $t : [0..1];$
INITIALLY $t = 1;$

PROCESS P_1

DECLARE $y_1 : [0..1];$
INITIALLY $y_1 = 0;$

L_0 : goto L_0; | $\{y_1 := 1;$ goto $L_1;\}$

$L_1 : t := 1;$

L_2 : if $(y_2 = 0 \vee t = 0)$ goto $L_3;$
if $(y_2 = 1 \wedge t = 1)$ goto $L_2;$

critical section $L_3 : y_1 := 0;$ goto $L_0;$
END

||

PROCESS P_2

DECLARE $y_2 : [0..1];$
INITIALLY $y_2 = 0;$

M_0 : goto M_0; | $\{y_2 := 1;$ goto $M_1;\}$

$M_1 : t := 0;$

M_2 : if $(y_1 = 0 \vee t = 1)$ goto $M_3;$
if $(y_1 = 1 \wedge t = 0)$ goto $M_2;$

critical section $M_3 : y_2 := 0;$ goto $M_0;$
END

t is a global variable and y_1 and y_2 are local variables of P_1 and P_2, respectively. All variables are of type integer with range [0..1]. All statements appearing under the same label are assumed to be one atomic statement, i.e. nothing is interleaved between them.

Since each process has four possible locations we can view each program counter as a variable with range [0..3]. To describe the program computations we transform each program variable into a set of propositions, standing for the bit representation of the variable. We use t, y_1 and y_2 as propositions to represent the corresponding variables. $P_{1_0}, P_{1_1}, P_{2_0}$ and P_{2_1} are used to represent the program counters of P_1 and P_2, respectively. In the initial state of the program, $t = 1, y_1 = 0, y_2 = 0$ and P_1 and P_2 are in L_0 and M_0, respectively. Hence, for all program computations, t is true and $y_1, y_2, P_{1_0}, P_{1_1}, P_{2_0}$ and P_{2_1} are false in the first state of the computation. From this initial state there are four possible transitions:

1. P_1 executes the statement "goto L_0", resulting in the same state.
2. P_1 executes the statement "$y_1 := 1$; goto L_1", resulting in a state where t, y_1 and P_{1_0} hold and y_2, P_{1_1}, P_{2_0} and P_{2_1} are false.
3. P_2 executes the statement "goto M_0", resulting in no change in state.
4. P_2 executes the statement "$y_2 := 1$; goto M_1", resulting in a state where t, y_2 and P_{2_0} hold and y_1, P_{1_0}, P_{1_1} and P_{2_1} are false.

The construction of the global transition graph of the program proceeds in this way. Since the number of variables and the range of each is finite, there are only a finite number of different states.

The safety property that we want to check for this algorithm is that it really guarantees mutual exclusion, i.e., it is never the case that P_1 is in L_3 and P_2 is in M_3 at the same time. Hence we want all program computations to satisfy

$$\Box\neg(P_{1_0} \wedge P_{1_1} \wedge P_{2_0} \wedge P_{2_1})$$

The liveness property is that each of the processes–if it is not idle forever (i.e., at L_0 or M_0)– eventually gets to execute its critical section. Hence, we want all program computations to satisfy

$$\Box((P_{1_0} \wedge \neg P_{1_1} \vee \neg P_{1_0} \wedge P_{1_1}) \rightarrow \Diamond(P_{1_0} \wedge P_{1_1}))$$

and

$$\Box((P_{2_0} \wedge \neg P_{2_1} \vee \neg P_{2_0} \wedge P_{2_1}) \rightarrow \Diamond(P_{2_0} \wedge P_{2_1}))$$

Often, we want to verify that the program satifies these properties under some *fairness* condition, to exclude those executions in which one of the processes is not active from some point on in the computation. For example, we may want to consider the above

properties under the assumption that each process executes infinitely many transitions. It is rather clear that Peterson's algorithm above does not satisfy the liveness property if we do not assume such fairness. To specify those computations that satisfy the required fairness condition, we use an additional proposition. The proposition p_1 will hold only in these states in which P_1 is active; namely, only P_1 can execute a transition from a state in which p_1 is true. Thus, we have now two different initial states, one in which p_1 holds and one in which $\neg p_1$ holds. Each of these states has four possible next states, resulting from the two transitions the active process can take and the two alternatives for the next active process. To specify the liveness property above, under fairness, we use the following:

$$\Box(\Diamond p_1 \wedge \Diamond(\neg p_1)) \rightarrow \Box((P_{1_0} \wedge \neg P_{1_1} \vee \neg P_{1_0} \wedge P_{1_1}) \rightarrow \Diamond(P_{1_0} \wedge P_{1_1}))$$

4 The tableau algorithm

The satisfiability problem for temporal logic formulae is NP-complete. In the worst case, the number of steps needed to decide if a given formula p is satisfiable is $O(2^{|p|})$. The *tableau* algorithm for checking the satisfiability of a linear temporal formula [PS81] is aimed at avoiding the exponential worst case, when possible, by generating only those states that are necessary. The algorithm consists of two parts:

1. Given a formula p, a directed graph $M_p = (N_p, E_p, r)$ is constructed, and a set $\pi(n)$ of atomic propositions is associated with each node n in N. This graph is "locally consistent" in the sense that for each node n, the set of formulae $\pi(n)$ is consistent for all formulae except for path formulae involving $\Diamond$ and $\mathcal{U}$.

2. Checking global consistency: for each node $n \in N_p$ and for each formula $\Diamond p, q\mathcal{U}p$ in $\pi(n)$, check if there exists a path from n that eventually satisfies q.

In the following, we assume (without loss of generality) that all paths in the model are infinite. We can assume that the formula to be checked is of the form $p \wedge \Box T$.

4.1 The construction part

We distinguish between two types of formulae called α and β formulae.

α formulae are those that can be expressed by a conjunction of their subformulae, e.g., $\Box p \equiv p \wedge \bigcirc\Box p$. In the construction procedure an α formula r is replaced by the set of its subformulae, denoted $\alpha(r)$, as follows :

r	$\alpha(r)$
$p \wedge q$	$\{p, q\}$
$\Box p$	$\{p, \bigcirc \Box p\}$
$\neg(p \vee q)$	$\{\neg p, \neg q\}$
$\neg(p \rightarrow q)$	$\{p, \neg q\}$
$\neg(\Diamond p)$	$\{\Box(\neg p)\}$
$\neg(\Box p)$	$\{\Diamond(\neg p)\}$

β formulae are those that can be expressed as a disjunction of their subformulae, e.g., $\Diamond p \equiv p \vee \bigcirc \Diamond p$. In the construction, procedure a node with a β formula r is replaced by two nodes, each containing one of the sets of subformulae, denoted $\beta_1(r), \beta_2(r)$ and defined as follows:

r	$\beta_1(r)$	$\beta_2(r)$
$p \vee q$	$\{p\}$	$\{q\}$
$p \rightarrow q$	$\{\neg p\}$	$\{q\}$
$p \equiv q$	$\{p, q\}$	$\{\neg p, \neg q\}$
$p \mathcal{U} q$	$\{q\}$	$\{p, \bigcirc(p \mathcal{U} q)\}$
$\Diamond p$	$\{p\}$	$\{\bigcirc \Diamond p\}$
$\neg(p \wedge q)$	$\{\neg p\}$	$\{\neg q\}$
$\neg(p \equiv q)$	$\{\neg p, q\}$	$\{p, \neg q\}$
$\neg(p \mathcal{U} q)$	$\{\Box(\neg q)\}$	$\{(\neg q \mathcal{U}(\neg p \wedge \neg q)\}$

To describe the construction algorithm, we define for a set of formulae ϕ:

The set **next**(ϕ) is the set of formulae that must be true in a successor of any state that satisfies all $p \in \phi$.

$$\textbf{next}(\phi) = \{q \mid \bigcirc q \in \phi\}$$

The set **basic**(ϕ) is the subset of ϕ that uniquely identifies a state satisfying all $p \in \phi$.

$$\textbf{basic}(\phi) = \{p \mid p \in \phi \text{ and } (p \text{ is atomic or } p = \bigcirc q)\}$$

The function **analyze**(ϕ), provides for a set of formulae ϕ, a set S of sets of formulae resulting from repeated applications of α, β rules to formulae in ϕ. The function **analyze** is used to construct $M_p = (N_p, E_p)$, as follows:

construct(p)
start with root $r, \pi(r) = \{p\}$, $N_p = \{r\}, E_p = \emptyset$.
$S =$**analyze**($\pi(r)$)
for $\phi \in S$
if **basic**(ϕ) = **basic**($\pi(n)$) for some $n \in N_p$

add (r, n) to E_p
else
add a new node n to N_p, with $\pi(n) = \phi$
add (r, n) to E_p
for $m \in N_p$ a leaf in (N_p, E_p)
$S =$ **analyze**(**next**$(\pi(m))$)
for $\phi \in S$
if **basic**$(\phi) =$ **basic**$(\pi(n))$ for some $n \in N_p$
add (m, n) to E_p
else
add a new node n to N_p, with $\pi(n) = \phi$
add (m, n) to E
end construct

The function **analyze** is formally defined as follows :

analyze(ϕ_0)
$X := \{(\phi_0, \emptyset)\}$
for $(\phi, \varphi) \in X$ with $\phi \neq \emptyset$
$X := X - \{(\phi, \varphi)\}$
if $p \in \phi$
if p is an α formula
$\phi := (\phi - \{p\}) \cup \{q \mid q \in \alpha(p)$ and $q \notin \varphi\}$
$\varphi := \varphi \cup \{p\}$
$X := X \cup \{(\phi, \varphi)\}$
if p is a β formula
$\phi_1 := (\phi - \{p\}) \cup \{q \mid q \in \beta_1(p)$ and $q \notin \varphi\}$
$\phi_2 := (\phi - \{p\}) \cup \{q \mid q \in \beta_2(p)$ and $q \notin \varphi\}$
$\varphi := \varphi \cup \{p\}$
$X := X \cup \{(\phi_1, \varphi), (\phi_2, \varphi)\}$
if p is an atomic formula or $p = \bigcirc q$
$\phi := \phi - \{p\}$
$\varphi := \varphi \cup \{p\}$
$X := X \cup \{(\phi, \varphi)\}$
$S := \emptyset$
for $(\emptyset, \varphi) \in X$
if there is no $q, \neg q \in \varphi$
$S := S \cup \{\varphi\}$
return (S)
end analyze

4.2 Checking eventualities

When the construction of the structure (N_p, E_p) is completed, each node in the graph is locally consistent, but we still have to check for global consistency. That is, we have to check that each node satisfies the following two properties:

1. the node has at least one successor.
2. for each formula of the form $\Diamond p$ or $q\ \mathcal{U} p$ in the set of formulae $\pi(n)$ of a node n, there exists a path leading from n to a node m, such that $p \in \pi(m)$.

Each node that does not satisfy these two properties is removed from the graph. The formula p is satisfiable if and only if the remaining graph contains the root node r.

[LP85] shows that when checking temporal consistency it is enough to check consistency in the strongly connected components of the graph.

A *strongly connected component* (SCC) in a directed graph is a maximal set of nodes of the graph such that there is a path between each pair of nodes in the set.

We say that a SCC (N', E') is *consistent* in a model M, if for every $n \in N'$ and every $\Diamond p, q\mathcal{U}p \in \pi(n)$ there exists $m \in N'$ with $p \in \pi(m)$, and if N' contains a single node n, then there exists some successor of n in M.

check(N_p, E_p) checks temporal consistency in the locally consistent model M_p:

check(N_p, E_p)
  find strongly connected components in (N_p, E_p).
  For G a leaf SCC in (N_p, E_p)
    if G is consistent
      stop
    else
      remove G from (N_p, E_p)
  p is *satisfiable* iff $r \in N_p$
end check

If p is satisfiable, then a model can be constructed from the the graph (N_p, E_p). For every node $n \in N_p$ and every $\Diamond p$ or $q\mathcal{U}p$ in $\pi(n)$ we know that there exists a path from n leading to a node m such that $p \in \pi(m)$. However, we have to construct *one* infinite path, starting from *root* such that for all nodes along the path, all formulae will be satisfied along this path. The following procedure, **build_model**, defines such an infinite path. Note that ; (the semicolon) here denotes concatenation.

build_model(N_p, E_p)
  $\delta(n)$ denotes the ordered list of successors of a node n.

```
    n := r
    σ := n
    while (true)
          m := head of δ(n)
          σ := σ; m
          δ(n) := tail(δ(n)); m
          n := m
end build_model
```

The tableau algorithm for checking satsifiability of a temporal formula p is composed of the following steps:

1. **construct**(p) to get the structure $M_p = (N_p, E_p)$
2. **check**(N_p, E_p)
3. if $r \in N_p$
   p is satisfiable
   else
   $\neg p$ *is valid*

In the worst case, the size of M_p is $O(2^{|p|})$.

5 The satisfiability approach vs. the model checking approach

Given a finite state program A, a property p and fairness condition F, our goal is to verify whether every fair (according to F) execution of the program satisfies p. Two basic approaches to this problem are described here. One is to construct a formula $\phi(A, F, p)$ consisting of the possible transitions that can be executed by the different processes, the fairness condition and the property p, such that $\neg\phi(A, F, p)$ is valid iff A satisfies p under F. The other approach is to construct the transition graph for the program A, and check if $F \rightarrow p$ is satisfied along every possible execution path in the transition graph.

5.1 Verifying by checking satisfiability

We are given a program A, composed of n processes $P_1, P_2, \ldots, P_n$, a property p, and fairness condition F; we wish to verify that any possible interleaving execution of the processes that satisfies F satisfies p.

To verify that A satisfies p under F:

1. For a given algorithm A, a property p and fairness condition F, construct the temporal formula $\phi(A, F, p)$.
2. Use the tableau algorithm to check satisfiability of $\phi(A, F, p)$.
3. A satisfies p under F if and only if the tableau algorithm terminates with $\neg\phi(A, F, p)$ *valid.*

$\phi(A, F, p)$ is of the form

$$\phi(A, F, p) = \neg((I \wedge \Box(\bigvee_{i=1}^{n} \theta(P_i))) \rightarrow (F \rightarrow p))$$

I specifies the initial state, and $\theta(P_i)$ specifies the transitions that can be executed by process P_i. If $\phi(A, F, p)$ is satisfiable, then there exists a model whose initial state satisfies I, each of the model transitions from one state to the next corresponds to one of the possible processes transitions specified by $\theta(P_i), i = 1, \ldots, n$, and the model satisfies the fairness condition F, but it does not satisfy p. This implies that this model describes a fair execution of the program that does not satisfy the required property; hence, A does not satisfy p. If $\phi(A, F, p)$ is not satisfiable it means that $\neg\phi(A, F, p)$ is valid, hence any model corresponding to a program execution specified by $I \wedge \Box(\vee_{i=1}^{n}\theta(P_i)))$ that is a fair execution (i.e., satisfies F) must also satisfy p, hence A satisfies p under F.

To be able to specify fairness, we use for each process a proposition p_i, which must hold in every state where process P_i is active. The general form of $\theta(P_i)$, is

$$p_i \wedge \mu(P_i) \wedge (\bigvee_{k=1}^{m} t_k)$$

where $\{t_k \mid k = 1, \ldots, m\}$ stand for the set of transitions that can be executed by P_i and $\mu(P_i)$ guarantees that when P_i is active it can not change the values of any variable that is a local variable of another process. $\mu(P_i)$ has the form

$$\bigwedge_{j \neq i} \bigwedge_{x \text{ variable of } P_j} (x \equiv \bigcirc x)$$

Note that the set of local variables of a process include the program counter for this process.

Each of the t_m is specified by a formula that is a conjunction of two parts. The first part specifies the current state, which includes the program counter value and a condition stating some values of some variables. The second part specifies changes of values in the next state resulting from executing the transition: change of program counter and change

of variables by an assignment statement. All other local and shared variables are not changed.

To demonstrate how $\phi(A, F, p)$ is constructed, we now define the parts of the formula for Peterson's mutual exclusion algorithm.

$$I \equiv \neg P_{1_0} \wedge \neg P_{1_1} \wedge \neg P_{2_0} \wedge \neg P_{2_1} \wedge t \wedge \neg y_1 \wedge \neg y_2$$

For process P_1:

$$\begin{aligned}
\mu(P_1) &\equiv (y_2 \equiv \bigcirc y_2) \wedge (P_{2_0} \equiv \bigcirc P_{2_0}) \wedge (P_{2_1} \equiv \bigcirc P_{2_1}) \\
t_1^1 &\equiv \neg P_{1_0} \wedge \neg P_{1_1} \wedge \\
&\quad (\bigcirc \neg P_{1_0} \wedge \bigcirc \neg P_{1_1} \wedge (y_1 \equiv \bigcirc y_1) \wedge (t \equiv \bigcirc t) \vee \\
&\quad \bigcirc P_{1_0} \wedge \bigcirc \neg P_{1_1} \wedge \bigcirc y_1 \wedge (t \equiv \bigcirc t)) \\
t_2^1 &\equiv P_{1_0} \wedge \neg P_{1_1} \wedge \\
&\quad \bigcirc \neg P_{1_0} \wedge \bigcirc P_{1_1} \wedge (y_1 \equiv \bigcirc y_1) \wedge \bigcirc t \\
t_3^1 &\equiv \neg P_{1_0} \wedge P_{1_1} \wedge (\neg y_2 \vee \neg t) \wedge \\
&\quad \bigcirc P_{1_0} \wedge \bigcirc P_{1_1} \wedge (y_1 \equiv \bigcirc y_1) \wedge (t \equiv \bigcirc t) \\
t_4^1 &\equiv \neg P_{1_0} \wedge P_{1_1} \wedge (y_2 \wedge t) \wedge \\
&\quad \bigcirc \neg P_{1_0} \wedge \bigcirc P_{1_1} \wedge (y_1 \equiv \bigcirc y_1) \wedge (t \equiv \bigcirc t) \\
t_5^1 &\equiv P_{1_0} \wedge P_{1_1} \wedge \\
&\quad \bigcirc \neg P_{1_0} \wedge \bigcirc \neg P_{1_1} \wedge \bigcirc \neg y_1 \wedge (t \equiv \bigcirc t)
\end{aligned}$$

Hence

$$\theta(P_1) \equiv p_1 \wedge \mu(P_1) \wedge (t_1^1 \vee t_2^1 \vee t_3^1 \vee t_4^1 \vee t_5^1)$$

For process P_2:

$$\begin{aligned}
\mu(P_2) &\equiv (y_1 \equiv \bigcirc y_1) \wedge (P_{1_0} \equiv \bigcirc P_{1_0}) \wedge (P_{1_1} \equiv \bigcirc P_{1_1}) \\
t_1^2 &\equiv \neg P_{2_0} \wedge \neg P_{2_1} \wedge \\
&\quad (\bigcirc \neg P_{2_0} \wedge \bigcirc \neg P_{2_1} \wedge (y_2 \equiv \bigcirc y_2) \wedge (t \equiv \bigcirc t) \vee \\
&\quad \bigcirc P_{2_0} \wedge \bigcirc \neg P_{2_1} \wedge \bigcirc y_2 \wedge (t \equiv \bigcirc t)) \\
t_2^2 &\equiv P_{2_0} \wedge \neg P_{2_1} \wedge \\
&\quad \bigcirc \neg P_{2_0} \wedge \bigcirc P_{2_1} \wedge (y_2 \equiv \bigcirc y_2) \wedge \bigcirc \neg t
\end{aligned}$$

$$
\begin{aligned}
t_3^2 &\equiv \neg P_{2_0} \wedge P_{2_1} \wedge (\neg y_1 \vee t) \wedge \\
&\quad \bigcirc P_{2_0} \wedge \bigcirc P_{2_1} \wedge (y_2 \equiv \bigcirc y_2) \wedge (t \equiv \bigcirc i) \\
t_4^2 &\equiv \neg P_{2_0} \wedge P_{2_1} \wedge (y_1 \wedge \neg t) \wedge \\
&\quad \bigcirc \neg P_{2_0} \wedge \bigcirc P_{2_1} \wedge (y_2 \equiv \bigcirc y_2) \wedge (t \equiv \bigcirc t) \\
t_5^2 &\equiv P_{2_0} \wedge P_{2_1} \wedge \\
&\quad \bigcirc \neg P_{2_0} \wedge \bigcirc \neg P_{2_1} \wedge \bigcirc \neg y_2 \wedge (t \equiv \bigcirc t)
\end{aligned}
$$

Hence

$$\theta(P_2) \equiv p_2 \wedge \mu(P_2) \wedge (t_1^2 \vee t_2^2 \vee t_3^2 \vee t_4^2 \vee t_5^2)$$

A fair execution is an execution in which each process is active infinitely often, specified by $\Box(\Diamond p_1 \wedge \Diamond p_2)$.

It follows that in order to check safety of the Peterson mutual exclusion algorithm under fairness, we have to check satisfiability of the formula

$$\neg((I \wedge \Box(\theta(P_1) \vee \theta(P_2))) \rightarrow (\Box(\Diamond p_1 \wedge \Diamond p_2) \rightarrow \Box(\neg(P_{1_0} \wedge P_{1_1} \wedge P_{2_0} \wedge P_{2_1}))))$$

and to check liveness

$$
\begin{aligned}
&\neg((I \wedge \Box(\theta(P_1) \vee \theta(P_2))) \rightarrow \\
&\quad (\Box(\Diamond p_1 \wedge \Diamond p_2) \rightarrow \Box((\neg P_{1_0} \wedge P_{1_1} \vee P_{1_0} \wedge \neg P_{1_1}) \rightarrow \Diamond(P_{1_0} \wedge P_{1_1}))))
\end{aligned}
$$

Since checking satisfiability is exponential in the size of the formula that is checked, using the formula $\phi(A, F, p)$ to check if A satisfies p may be, in the worst case, exponential in the length of the program plus the size of the property.

5.2 Verifying by model checking

In the model checking approach, the transition graph of the program is constructed and then it is checked to determine if there exists a path in the graph that satisfies $\neg(F \rightarrow p)$

1. Construct the global transition graph $(S, T)_A$ for the program A.
2. Use **construct**$(\neg(F \rightarrow p))$ to construct a structure $M_{\neg(F \rightarrow p)} = (N, E)$.
3. Cross product $M_{\neg(F \rightarrow p)}$ and $(S, T)_A$ to get a locally consistent structure, denoted $(V, R)_{(A,F,p)}$, with a root node denoted v_0.
4. **check**$((V, R)_{(A,F,p)})$
5. A satisfies p under F if and only if $v_0 \notin V$.

If $v_0 \in V$ after **check**$((V,R)_{(A,F,p)})$, then the remaining structure contains a path corresponding to a possible execution of the program A, and this path satisfies the formula at the root of $(N,E)_{\neg(F \rightarrow p)}$, that is, it satisfies $\neg(F \rightarrow p)$. Hence, there exists an execution that satisfies F but does not satisfy p. This implies that A does not satisfy p under F. If v_0 is deleted from V, then there does not exist a path in the transition graph that satisfies $\neg(F \rightarrow p)$, hence every execution of the program A that satisfies F satisfies p. This implies that A satisfies p under F.

5.2.1 Constructing the global transition graph

The global transition graph consists of the set of nodes S and the set of edges T. Each $s \in S$ stands for a program state and a set $\pi(s)$ of propositions and negations of propositions is associated with it. The set $\pi(s)$ provides the value of the program variables in the state s, by the corresponding propositions. In each state s, exactly one process P_i is identified as active, by setting the corresponding p_i to true and p_j for all $j \neq i$ to false. The construction of the graph starts with a special root state s_0 such that $\pi(s_0)$ corresponds to the values of program variables at the initial state. s_0 has n successors, where successor i corresponds to the initial state with process i active, i.e., p_i is true. The construction of the transition graph proceeds by generating transitions from each state, according to the statements that can be executed from the state by the active process, and creating states corresponding to the resulting program state with one of the processes being the next active process. Since the program is finite state, the transition graph construction must terminate.

5.2.2 Cross product of the two structures

The cross product procedure is defined by taking those pairs (n,s), $n \in N$ and $s \in S$, which are consistent. A pair (n,s) is *consistent* if and only if for every atomic formula P, at most one of $P, \neg P$ is in $\pi(n) \cup \pi(s)$. V is initialized to v_0 corresponding to the pair (r,s_0), that is, $\pi(v_0) = \pi(r) \cup \pi(s_0)$. R is initialized to the empty set. $(V,R)_{(A,F,p)}$ is constructed by **cross_product**.

cross_product$((N,E,r),(S,T,s_0))$
start with root $v_0, \pi(v_0) := \pi(r) \cup \pi(s_0), V := \{v_0\}, E := \emptyset$
for $v \in V$ a leaf in (V,R)
for $m \in N, (n,m) \in E$
for $t \in S, (s,t) \in T$
if (m,t) is consistent
if for some $u \in V$ **basic**(u) =**basic**$(\pi(m)) \cup \pi(t))$
add (v,u) to R
else

add a new node u to V with $\pi(u) = \pi(m) \cup \pi(t)$
add (v, u) to R
end cross_product

6 Implementation and experimental results

6.1 Implementation of the tableau algorithm

Several issues seemed to be vital to an efficient implementation of the tableau algorithm.

6.1.1 The form of the LTL formula

1. Conjunctive form has an advantage over disjunctive form. A *disjunctive* form of a subformula specifying a transition in the program is: $p_s \wedge c_s \rightarrow \bigcirc q_{s'}$ where p_s and c_s specify the current program counter and a condition, respectively and $q_{s'}$ specifies the next state. The program is specified by the conjunction of all disjunctive formulae specifying transitions. An equivalent specification for the program is by disjunction of all transitions specified in *conjunctive* form by $p_s \wedge c_s \wedge \bigcirc q_{s'}$ (see the formula in Section 5.1). The implementation of the tableau algorithm was faster for the conjunctive form, probably because of the priority we give to rules applied to α formulae (see Section 6.1.3.).

2. Putting the "next" operator ($\bigcirc$) as "low" as possible turned out to be more efficient than an equivalent form in which next operators appeared before $\wedge$, $\vee$ or $\neg$ operators.

6.1.2 Representation of sets of formulae

Each node in the constructed graph is associated with a set of formulae. The set of all possible subformulae of the formula to be checked is computed (the size of the set is at most $5|p|$ for a formua p) and the formulae are numbered. A set is represented by an array of bits, where the ith bit is 1 iff the ith formula is in the set. This representation allows operations such as union, checking consistency of a set, checking membership, etc. to be performed efficiently by bit operations.

6.1.3 Application of α, β rules

In the construction procedure, the same rule may be applied to the same formula many times. To avoid some of this duplication, α rules should have prioprity over β rules, as the following shows. Consider $\phi = \{q_1, q_2, \ldots, q_k\}$, a set of formulae where q_1 is an α formula and q_2 is a β formula. If we first apply the β rules to q_2, we have two nodes

associated with $\phi_1 = \{q_1, \beta_1(q_2), \ldots, q_k\}$ and $\phi_2 = \{q_1, \beta_2(q_2), \ldots, q_k\}$. Thus, the rules for $q_1, q_3, \ldots, q_k$ must be applied at least twice. Applying the α rule for q_1 in ϕ will result in a set $\{\alpha(q_1), q_2, \ldots, q_k\}$.

To try to minimize duplicate applications of the same rule to the same formula, we construct, in a preprocessing procedure, a table T, providing the results of repeated applications of α rules to subformulae. The set of subformulae $CL(p)$ is computed. For each $q \in CL(p)$, the set $\alpha(q)$ or the sets $\beta_1(q), \beta_2(q)$ resulting from applying the corresponding rule for q are stored in $T[q]$, the entry corresponding to q in the table T. Then, the α closure of T is computed by repeatedly applying α rules to each set ϕ in the entry $T[q]$ in the table. A set ϕ is α *closed* if for every α formula $q \in \phi, \alpha(q) \subseteq \phi$. α rules are applied to sets in the table until each set ϕ is α closed. In the construction algorithm, a rule is applied to formula q in a set ϕ by replacing ϕ by $\phi \cup \psi$ for each ψ in $T[q]$.

6.1.4 Search of nodes in the graph

With the generation of a set ϕ the graph is searched to find if there exists a node n such that **basic**$(\pi(n))$ = **basic**(ϕ). To enable an efficient search we interpret the binary representation of a set $\pi(n)$ as a number that is used as a unique id of the node n. The ids are hashed (see [K73]) into a table of size K, such that on the average entry i in the table will have a list of $|N_p|/K$ nodes. Hence, to search the graph for the set **basic**(ϕ), we search the nodes in the entry corresponding to **basic**(ϕ) in the hash table.

6.2 Implementation of the model checker

The implementation of the construction of the global transition graph uses ideas similar to those described above. States are represented by the set of propositions and negation of propositions that hold in the state. When the cross product with the model M_p is computed the resulting structure is composed of nodes each of which is a pair of pointers (p_s, p_n) where p_s is pointer to a state in the global transition graph and p_n is a pointer to a node in the model. The set associated with (p_s, p_n) is $\pi(s) \cup \pi(n)$.

6.3 Experimental results

In the current implementation the model checking approach was up to 10 times faster than the satisfiability approach. The following results were measured on a SUN 4 for two mutual exclusion algorithms X_1 and X_2 (see Appendix) X_1 is a mutual exclusion algorithm for n processors that guarantees safety and *communal liveness* but not liveness. *Communal liveness*, in this case, means that if *some* process is asking to use the resource, then eventually *some* (not necessarily the same) process will get to execute its critical

section. Algorithm X_2 is a more complicated mutual exclusion algorithm. It guarantees safety and liveness for n processors.

Each row in the table provides execution time in seconds and the number of nodes in the locally consistent structure. The results relate to the verification of the program composed of n processes of algorithm X_1 or X_2 (in the first column) when checked against the *property* in the second column.

Note that some results are missing for the model checking program. This is because the current implementation (to be corrected in the near future) of the compiler requires a lot of manual preparation.

Algorithm	*Property*	*Satisfiability*		*Model Checking*	
		Nodes	*Time*	*Nodes*	*Time*
$X_1, n = 4$	Safety	1514	18.67	1189	5.61
$X_1, n = 4$	Liveness	2423	78.94	2566	8.75
$X_1, n = 4$	Com Liveness	2171	78.39	3493	55.08
$X_1, n = 5$	Safety	6752	113.37		
$X_1, n = 5$	Liveness	10289	582.03		
$X_1, n = 5$	Com Liveness	9587	560.51		
$X_1, n = 6$	Safety	28190	698.46		
$X_1, n = 6$	Liveness	91316	8125.0		
$X_1, n = 6$	Com Liveness	79436	7616.0		
$X_2, n = 4$	Safety	4598	214.46	3429	50.49
$X_2, n = 4$	Liveness	6824	885.02	7541	69.53
$X_2, n = 5$	Safety	23012	2106.17		
$X_2, n = 5$	Liveness	73330	19742.0		

7 Conclusions

The results presented here indicate that model checking for LTL may indeed be implemented efficiently.

In [CS89], a distributed implementation of the satisfiability algorithm, provides further improvement.

Our model checker provides the user with

- The expressive power of Linear Temporal Logic.
- The ability to check, given two LTL specifications, whether one implies the other.
- Efficiency comparable to the CTL model checker.

Acknowledgments

This report is dedicated to Dr. H. Schorr and all others who do not believe that verification is important and/or useful.

David Mizell made this work possible.

Susan Coatney implemented the compiler in the best possible way.

Danny Cohen and Jon Postel believed that this report was worth writing.

References

[BMP83] M. Ben-Ari, Z. Manna, A. Pnueli, "The temporal logic of branching time," *Acta Informatica* 20, (1983), pp. 207-226.

[B86] M.C. Browne, "An improved algorithm for the automatic verification of finite state systems using temporal logic," *Proceedings of the 1986 Conference on Logic in Computer Science*, 1986.

[BC86] M.C. Browne, E.M. Clarke, "SML: a high level language for the design and verification of finite state machines," *IFIP WG 10.2 International Working Conference from HDL Descriptions to Guaranteed Correct Circuit Designs*, 1986.

[CES83] E.M. Clarke, E.A. Emerson, A.P. Sistla, "Automatic verification of finite state concurrent systems using temporal logic specifications: a practical approach," 10^{th} *ACM Symposium on Principles of Programming Languages*, 1983.

[CG87] J.M. Crawford, D.M. Goldschlag, "The mechanical verification of distributed systems," Technical Report no. 7, Computational Logic Inc, 1987.

[CS89] K.M. Chandy, R. Sherman, "Parallel model checking," Technical Report, California Institute of Technology, 1989.

[EH86] E.A. Emerson, J. Y. Halpern, "Sometimes and not never revisited: on branching versus linear time temporal logic," *JACM Vol 33 No. 1*, January 1986.

[EL85] E.A. Emerson, C. Lei, "Modalities of model checking: branching time strikes back," 12^{th} *ACM Symposium on Principles of Programming Languages*, 1985.

[K73] D.E. Knuth, *Sorting and Searching*, Addison-Wesley, Reading, Mass, 1973.

[L80] L. Lamport, "Sometime is sometimes not never - on the temporal logic of programs," 7^{th} *ACM Symposium on Principles of Programming Languages*, 1980.

[LP85] O. Lichtenstein, A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification," 12^{th} *ACM Symposium on Prociples of Programming Languages*, 1985.

[P77] A. Pnueli, "The temporal logic of programs," 19^{th} *ACM Symposium on Foundations of Computer Science*, 1977.

[PS81] A. Pnueli, R. Sherman, "Semantic Ttbleau for temporal logic," Technical Report, Weizmann Institute, CS81 - 21, 1981.

[SC82] A.P. Sistla, E.M. Clarke, "The complexity of propositional temporal logic," 14^{th} *ACM Symposium on Theory of Computing*, 1982.

Appendix

Algorithm X_1 for 2 processes

```
DECLARE       y:[0..1];
INITIALLY     y=1;

PROCESS P1
   DECLARE    t1:[0..1];
   INITIALLY  t1=0;

      L0 : goto L0; | { t1 := 0; goto L1; }

      L1 : if (t1=0) goto L2;
           if (t1=1) goto L3;

      L2 : t1 :=: y; goto L1;

      // L3 - critical section
      L3 : t1 :=: y; goto L0;

END

||


PROCESS P2
  DECLARE     t2:[0..1];
  INITIALLY   t2=0;

     M0 : goto M0; | { t2 := 0; goto M1; }

     M1 : if (t2=0) goto M2;
          if (t2=1) goto M3;

     M2 : t2 :=: y; goto M1;

     // M3 - critical section
     M3 : t2 :=: y; goto M0;

END
```

Algorithm X_2 for 2 processes

```
DECLARE       y:[0..3];
INITIALLY     y=0;

PROCESS P1
   DECLARE    r1:[0..3]; t1:[0..1];
   INITIALLY  r1=3; t1=0;

      L0 : goto L0; | { t1 := 1; goto L1; }

      L1 : if (r1=3 ^  (y=0 | y=1)) goto L2;
           if (r1=3 ^ ~(y=0 | y=1)) goto L1;
           if (~(r1=3)) goto L2;

      L2 : if (~(y=0) ^ ~(y=1)) {r1:=:y; goto L1;}
           if ( (y=0) |  (y=1)) {r1:=:y; goto L3;}

      // L3 - critical section
      L3 : { r1 := 3; t1:=0; goto L4; }

      L4 : if (t2=1) {y:=2; goto L0;}
           if (t2=0) {      goto L5;}

      L5 : if (t1=1) {y:=1; goto L0;}
           if (t1=0) {y:=0; goto L0;}

END

||

PROCESS P2
   DECLARE    r2:[0..3]; t2:[0..1];
   INITIALLY  r2=3; t2=0;

      M0 : goto M0; | { t2 := 1; goto M1; }

      M1 : if (r2=3 ^  (y=0 | y=2)) goto M2;
           if (r2=3 ^ ~(y=0 | y=2)) goto M1;
           if (~(r2=3)) goto M2;
```

```
        M2 : if (~(y=0) ^ ~(y=2)) {r2:=:y; goto M1;}
             if ( (y=0) |  (y=2)) {r2:=:y; goto M3;}

        // M3 - critical section
        M3 : { r2 := 3; t2:=0; goto M4; }

        M4 : if (t1=1) {y:=1; goto M0;}
             if (t1=0) {      goto M5;}

        M5 : if (t2=1) {y:=2; goto M0;}
             if (t2=0) {y:=0; goto M0;}

END
```